SICS/T-91/9119

Efficient Algorithms for Computing Transitive Closure in CWB: Implementation and Comparison of Several Variations

Monica Dahlberg

SICS technical report
T91:19
ISSN 1100-3154

SICS/T-91/9119

Efficient Algorithms for Computing Transitive Closure in CWB: Implementation and Comparison of Several Variations

Monica Dahlberg

September 23, 1991

SICS technical report T91:19
ISSN 1100-3154

Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA
Sweden

Abstract

This paper presents a thesis work for the M.Sc. degree in Computer Science at the University of Stockholm. The work was accomplished at the Swedish Institute of Computer Science (SICS) during autumn and winter of 1989 - 1990. Björn Lisper was supervisor at SICS and Yngve Sundblad at NADA/KTH.

The task was to implement an efficient algorithm for computing the transitive closure of binary relations in the Concurrency Workbench (CWB).

The CWB is a system that caters for the analysis of concurrent processes expressed in CCS. It was developed at the University of Edinburgh. Further development has and is being carried out both at Edinburgh, the University of Sussex and SICS. The system is written in Standard ML.

I have studied a number of transitive closure algorithms. An algorithm by J.Eve and R.Kurki-Suonio seemed to be the most efficient. Two versions of this and of the classical Warshall algorithm have been implemented. One version uses array representation and one version uses list representation.

Sammanfattning

Denna rapport beskriver ett examensarbete i datalogi vid matematikerlinjen, Stochholms Universitet. Arbetet har utförts vid SICS (Swedish Institute of Computer Science) under hösten och vintern 1989 - 1990. Handledare på SICS var Björn Lisper och på NADA/KTH Yngve Sundblad.

Uppgiften bestod av att implementera en effektiv algoritm för att bilda transitiv slutning av binära relationer i Concurrency Workbench (CWB).

CWB är ett system som analyserar beskrivningar i CCS av distribuerade system. Det har utvecklats vid Universitetet i Edinburgh och vidareutvecklas nu både i Edinburgh, vid Universitetet i Sussex samt på SICS. Systemet är skrivet i Standard ML.

Jag har studerat ett antal olika algoritmer som beräknar transitiv slutning. En algoritm av J.Eve och R.Kurki-Suonio verkade vara mest intressant. Två versioner av vardera denna samt Warshall's klassiska algoritm har implementerats. I den ena versionen används array-representation och i den andra listrepresentation.

Contents

1 Introduction

This paper presents a thesis work for the M.Sc. degree in Computer Science at the University of Stockholm. The work was accomplished at the Swedish Institute of Computer Science (SICS) during autumn and winter of 1989 - 1990. Björn Lisper was supervisor at SICS and Yngve Sundblad at NADA/KTH.

The task was to implement an efficient algorithm for computing the transitive closure of binary relations in the Concurrency Workbench (CWB) [1, 2].

The CWB is an automated tool for verification and manipulation of concurrent finite-state processes expressed in Milner's Calculus of Communication Systems (CCS) [13].

The CWB system is written in Standard ML and it was developed at the University of Edinburgh. Further development has and is being carried out both at Edinburgh, the University of Sussex and SICS. This work was performed on version 5.0.

The CWB has been applied to examples involving the verification of communication protocols by, among others, Ellemtel and Swedish Telecom Radio. It has also been used in mutual exclusion algorithms and has proven to be a valuable aid in teaching and research.

The function transcl in the CWB has been identified as a particularly time consuming process. It is used in the minimize command, which is a frequently used command. The transcl function is a graph transforming function. It replaces $\xrightarrow{\tau}$ with the transitive closure of $\xrightarrow{\tau}$.

2 Transitive closure

The transitive closure of a relation R is the relation R^+ defined by aR^+b if and only if there is a sequence $c_1Rc_2, c_2Rc_3, \ldots, c_{m-1}Rc_m$ where $m \geq 2$, $a = c_1$, and $b = c_m$. It can easily be showed that $R^+ = \bigcup_{i=1}^{\infty} R^i$ and if the set has n elements then $R^+ = \bigcup_{i=1}^{n} R^i$. It requires $O(n^4)$ operations to compute R^+ this way.

A binary relation R over a set can be seen as a directed graph, where the elements of the set are vertices and there is an edge from vertex i to vertex j iff iRj. Forming the transitive closure of a directed graph means finding all directed paths in the graph.

Figure 1 shows the directed graph $G = \langle V, E \rangle$, where V is the set of vertices $\{1, 2, 3, 4, 5, 6\}$ and E is the set of edges $\{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 6, 2 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 4\}$. The transitive closure of G is seen in figure 2.

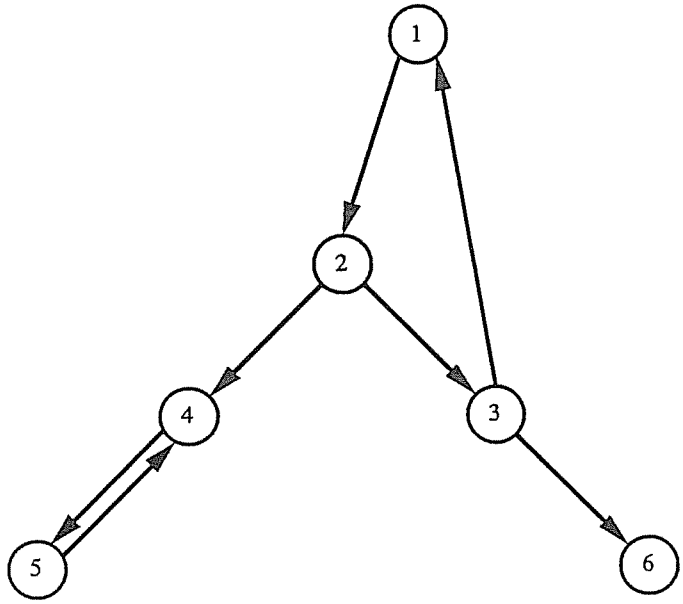

Figure 1: A directed graph G.

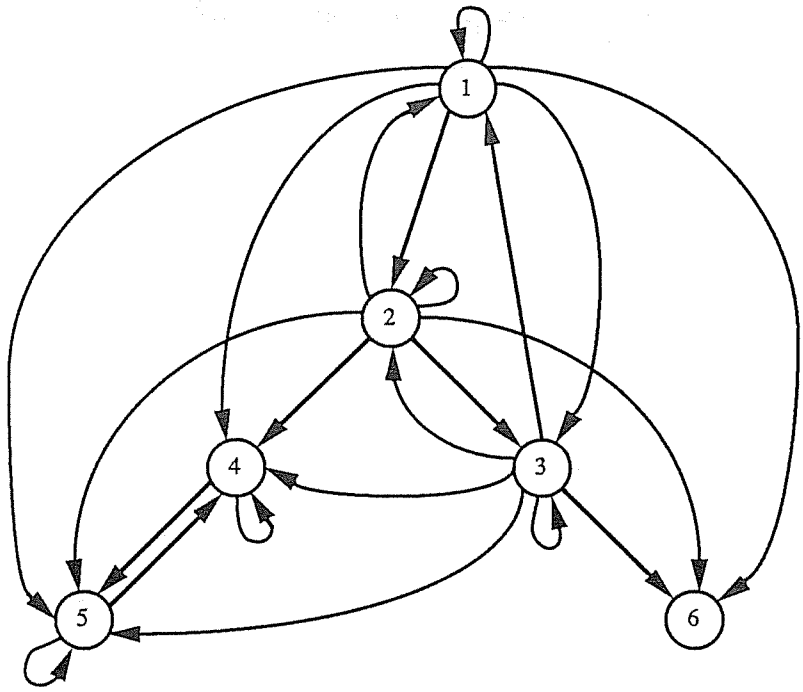

Figure 2: The transitive closure T of G.

In the CWB application vertices are replaced by states and edges by τ-transitions. The current graphs of interest are very sparse and large, about 1000 states is not uncommon.

There are a multitude of transitive closure algorithms se for instance [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14]. Some of these algorithms are utilizing the concept of *strongly connected components* (scc).

A scc of a directed graph is defined by a pair of sets $(X_i, \bar{X}_i)$. Where X_i is a subset of V, such that if x and y belong to X_i then there is both a path from x to y and a path from y to x. The members of $\bar{X}_i$ are the *scc edges* of X_i. The *head* of X_i is the least vertex in the set X_i. Figure 3 shows the strongly connected components of G, the head vertices are 1, 4, 6.

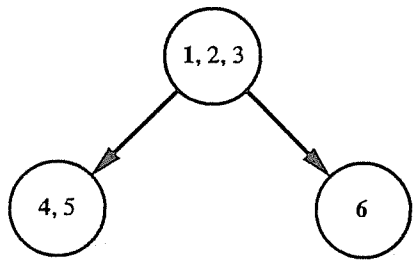

Figure 3: The strongly connected components of G.

I have studied and implemented two of the above referred algorithms, Warshall's [14, 5] and Eve and Kurki-Suonio's [4].

Warshall's algorithm is the simplest and in a computational experiment [10] it was found to be the fastest, among the compared algorithms, for very sparse graphs.

Eve and Kurki-Suonio's algorithm, which is the latest, also seemed to have interesting properties:

- single traversal of the graph
- utilizes the presence of strongly connected components
- demands only two extra array variables
- brief and easily understood
- easy to implement

3 Warshall's algorithm

The first description of a transitive closure algorithm is due to Warshall [14, 5]. It is simple and well-known. Below it is given with a $n \times n$ boolean matrix $m[i, j]$ representing the graph.

```
for 1 ≤ i ≤ n do
  for 1 ≤ r ≤ n do
    if m[r, i] then for 1 ≤ k ≤ n do
      m[r, k] := m[r, k] or m[i, k]
```

Where initially

$$m[i,j] = \begin{cases} true & \text{if there is an edge from vertex } i \text{ to vertex } j \\ false & \text{otherwise} \end{cases}$$

and after completion

$$m[i,j] = \begin{cases} true & \text{if there is a path from vertex } i \text{ to vertex } j \\ false & \text{otherwise} \end{cases}$$

The algorithm takes between $O(n^2)$ and $O(n^3)$ operations. Two versions, *wa* and *wl*, of this algorithm have been implemented. They differ in the way the edges of the graph are represented during the execution of the transcl function. The former uses a boolean array: *tauarr*, in a similar way as the original version. The latter uses a state list: *taus*, which should be more efficient for sparse relations. The complete source code of the transcl function is given in appendix A.1 and A.2.

4 Eve and Kurki-Suonio's algorithm

4.1 Introduction

An algorithm which combines a single traversal with the improved efficiency in the presence of strongly connected components is given by J. Eve and R. Kurki-Suonio [4]. It is based upon Tarjan's algorithm [11] for finding scc of a directed graph.

Each vertex of the graph is visited only once and each edge is inspected only once. The vertices in figure 1 will be visited in the following order: 1, 2, 3, 6, 4, 5. The vertices are pushed onto a stack as they are first encountered. When a scc is completely examined:

- all the vertices of this scc are removed from the stack
- their index values are changed to the same as the head vertex
- the head vertex is pushed into a list of head vertices

After performing this main part of the algorithm (see figure 5).

- all the scc of the graph will be identified, e. g. a list of head vertices is built and it is determined which scc each vertex belongs to
- all the edges from the head vertices are found

4.2 Type of edges

The edges in the finally transformed graph can be divided into four groups depending upon how the algorithm finds the edge.

A) Type A edges: edges in the original graph.
Edges of type A in T: $\{1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 6, 2 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 4\}$.

B) Type B edges: if the currently examined vertex x has an edge to a head of a previously examined scc, then add to the set of edges from x all the edges from that head vertex.
Edges of type B in T: $\{2 \rightarrow 5\}$

C) Type C edges: if the currently examined vertex x is a head of a scc, then add to the set of edges from x all the edges from all the other vertices in that scc.
Edges of type C in T: $\{4 \rightarrow 4, 1 \rightarrow 1, 1 \rightarrow 6, 1 \rightarrow 3, 1 \rightarrow 4, 1 \rightarrow 5\}$

D) Type D edges: all the other edges.
Edges of type D in T: $\{2 \rightarrow 1, 2 \rightarrow 2, 2 \rightarrow 6, 3 \rightarrow 2, 3 \rightarrow 3, 3 \rightarrow 4, 3 \rightarrow 5, 5 \rightarrow 5\}$

4.3 Implementation of the algorithm

The algorithm is reproduced below in basically the same form as Eve and Kurki-Suonio have described it:

```
(1)                 procedure transclosure
(2)                   begin integer stacktop,listindex,i;
(3)                   integer array VERTICES, INDEX [1::n];
(4)
(5)                   procedure closure (integer value x);
(6)                     begin integer w;
(7)                     INDEX[x]:= stacktop:= stacktop+1; VERTICES[stacktop]:= x;
(8)                     for w ∈ SONS(x) do begin
(9) type A edge           RPLUS(x,w):= true;
(10)                      if INDEX[w] = 0 then closure(w);
(11)                      if INDEX[w] ≤ stacktop then INDEX[x]:= min(INDEX[x],INDEX[w])
(12) type B edges         else RPLUS(x,*):= RPLUS(x,*) or RPLUS(VERTICES[INDEX[w]],*);
(13)                    end;
(14)                    if x = VERTICES[INDEX[x]] then begin
(15)                      listindex:= listindex-1; w:= VERTICES[stacktop];
(16)                      stacktop:= stacktop-1; INDEX[w]:= listindex;
(17)                      while w ≠ x do begin
(18) type C edges           RPLUS(x,*):= RPLUS(x,*) or RPLUS(w,*);
(19)                        w:= VERTICES[stacktop];
(20)                        stacktop:= stacktop-1; INDEX[w]:= listindex;
(21)                      end;
(22)                      VERTICES[listindex]:= x;
(23)                    end;
(24)                  end;
(25)
(26)                  listindex:= v+1; stacktop:= 0;
(27)                  for i:= 1 until n do INDEX[i]:= 0;
(28)                  for i:= 1 until n do if INDEX[i] = 0 then closure(i)
(29)                  for i:= 1 until n do
(30)                    if i ≠ VERTICES[INDEX[i]]
(31) type D edges       then RPLUS(i,*):= RPLUS(i,*) or RPLUS(VERTICES[INDEX[i]]);
(32)                end;
```

SONS(x) means the set of original sons to x. RPLUS is a two dimensional array, where all the elements are initialized to false and after completion

$$RPLUS(x,w) = \begin{cases} true & \text{if there is a path from vertex } x \text{ to vertex } w \\ false & \text{otherwise} \end{cases}$$

The stack is implemented in elements $1, 2, \ldots$ of the array VERTICES and the variable stacktop points to the top of the stack. After completion a list of head vertices will be built in elements $n, n-1, \ldots$ of the same array. The variable listindex points to the top of the list of head vertices. INDEX[i] contains the address to the element of VERTICES holding vertex i. After completion INDEX[i] will contain the address to the element of VERTICES holding the head of the scc that i belongs to: see figure 4.

4.4 Comments to the implementation

(26) initiate listindex and stacktop, (v is the number of vertices)

(27) initiate the INDEX array to zero
(zero means that the vertex have not been previosly examined)

(28) each vertex of the graph will be examined once and the examinatio will be done in increasing number order

(7) the currently examined vertex x is pushed onto the stack

(9) a type A edge is found

(10) examine the son vertex, if not previously done

(11) if the son vertex w is on the stack, (this means that x and w belongs to the same scc) give x the same index value as the head of this scc

(12) type B edges are found

(14) if x is a head of a scc then

(15) update the pointer to the list of head vertices

(15-21) remove all the vertices of this scc from the stack and change their index value to the same as the head vertex

(18) type C edges are found

(22) put x into the list of head vertices

(29-30) finally add all type D edges to the graph

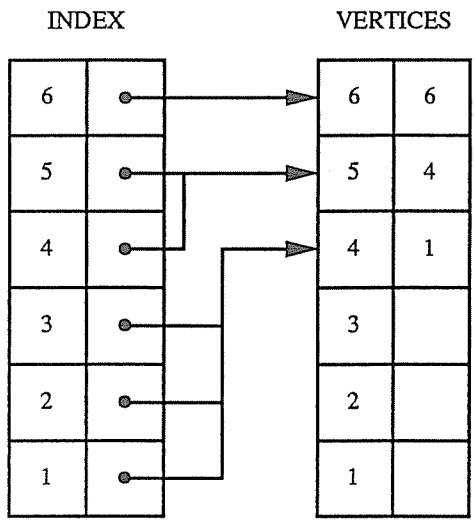

Figure 4: The contents of $VERTICES$ and $INDEX$ after completion.

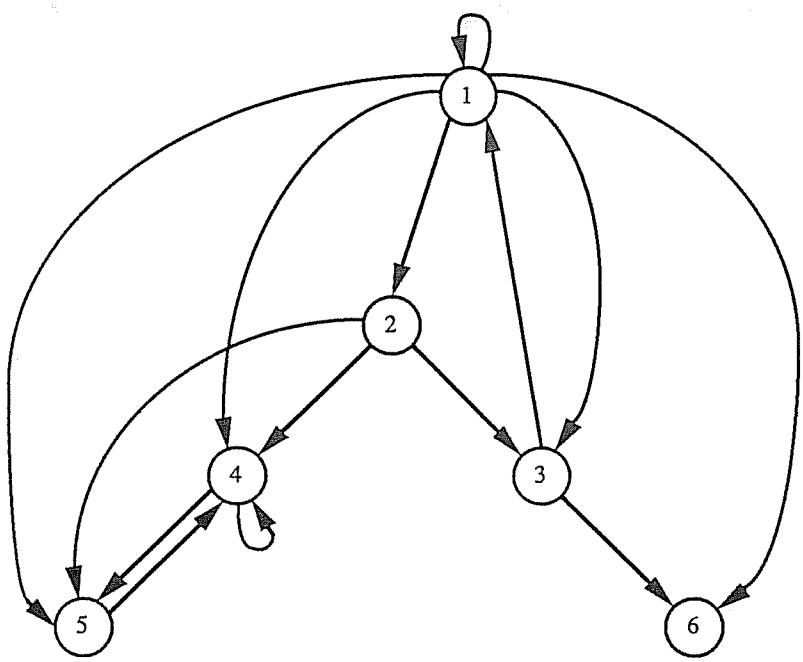

Figure 5: The graph after completion of the first part of the algorithm.

4.5 Example

The edges in the transitive closure of G, se figure 2, will be found in the following order:

- First all the original edges are found: $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 1$, $3 \rightarrow 6$, $2 \rightarrow 4$, $4 \rightarrow 5$, $5 \rightarrow 4$.
- When examining the head vertex 4, one type C edge from this vertex is added: $4 \rightarrow 4$.
- When examining the edge $2 \rightarrow 4$, one type B edge from vertex 2 is added: $2 \rightarrow 5$.
- When examining the head vertex 1, type C edges from this vertex are added: $1 \rightarrow 1$, $1 \rightarrow 6$, $1 \rightarrow 3$, $1 \rightarrow 4$, $1 \rightarrow 5$.
- Finally all type D edges are added: $2 \rightarrow 1$, $2 \rightarrow 2$, $2 \rightarrow 6$, $3 \rightarrow 2$, $3 \rightarrow 3$, $3 \rightarrow 4$, $3 \rightarrow 5$, $5 \rightarrow 5$.

5 Test results

Tests with the four new versions: *wa, ea, wl, el* and the *old* version have been performed on a 16 Mb sun 3/110 and on an 8 Mb sun 4/60. The operating system running on these machines was the SUN-OS 4.0.X and a New Jersey SML compiler was used.

Five CCS-agents from a communication protocol were used as test examples. The state space size for these agents ranges between 88 and 897 states. Table 1 shows the number of states and number of τ-transitions for these examples.

	User-Timers	PE	PE-Respnet	Net-Timers	Pro
size of state space	88	100	100	203	897
nr. of τ-transitions[1]	24	126	336	66	1373
nr. of τ-transitions[2]	27	148	929	96	21408

Table 1: Agents

The transcl function is used in the *minimize* command. Time measurements were made on the function transcl and on the entire minimize command. At the same time the opportunity was taken to check the times on the other major functions within the minimize command. These other functions are: *mkgraph, obscl, actcl, reflexcl* and *Equi.minimize*. They are related to each other in the following way: The minimize command consists essentially of the three major functions mkgraph, obscl and Equiv.minimize. Further, the function obscl is divided into three functions transcl, actcl and reflexcl.

Appendices C and D contain all test results. Every test has been repeated five times, but only the mean values are reported. The measured figures are of two kinds: *non gc time* (non garbage collection time) and *gc time* (garbage collection time). The *total time* (non gc time + gc time) is also displayed. All the reported figures are in seconds.

[1] before transcl
[2] after transcl

6 Conclusions

	User-Timers	PE	PE-Respnet	Net-Timers	Pro
old	0.5900	0.7620	1.4680	4.4340	206.2119
wa	0.3580	0.6100	2.0259	1.9899	232.5960
ea	0.1760	0.4240	0.8740	1.0680	48.6980
wl	0.1440	0.2260	0.4680	0.7800	27.5880
el	0.0280	0.0500	0.1240	0.0720	0.9780

Table 2: Execution time for the function transcl on sun 4/60.

	User-Timers	PE	PE-Respnet	Net-Timers	Pro
old	15.7240	19.7820	25.7220	88.4219	678.7639
wa	15.4759	19.4979	26.0160	86.2960	704.2580
ea	15.2119	19.1780	24.5679	84.5460	520.9819
wl	15.2800	17.2419	26.5259	84.9099	494.3460
el	15.1259	16.8759	27.4059	84.1700	476.3659

Table 3: Execution time for the minimize command on sun 4/60.

The results in table 2, of the measurements on the transcl function, show that el is the fastest version. It is considerably faster than the original version, never requiring more than 1/10 th of the original version's execution completion time. The time reduction for the entire minimize command is not as large. Table 3 shows that the execution completion time on the "Pro" agent is approximately 2/3 rds the original.

	old	el
mkgraph	24	34
transcl	30	0.2
actcl	44	64
Equiv.mi	2	2

Table 4: Percentage of execution time spent in each function.

Table 4 shows that the main part of the execution time in the original version is distributed in the following way: actcl 44% , transcl 30% and mkgraph 24%. With the new faster version (el) only 0.2% of the execution time is spent in the transcl function.

References

[1] R. Cleveland, J. Parrow, and B. Steffen. The concurrency workbench: Operating instructions, September 1988. University of Edinburgh, Laboratory for Foundations of Computer Science, Technical Note 10.

[2] R. Cleveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In *Proc. 9th Int. Symp. on Protocol Specification, Testing and Verification*, 1989. To appear.

[3] J. Dzikiewicz. An algorithm for finding the transitive closure of a digraph. *Computing*, 15:75–79, 1975.

[4] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.

[5] R. W. Floyd. Algorithm 96: Ancestor. *Comm. ACM*, 5:344–345, 1962.

[6] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1:56–58, 1971.

[7] P. O'Neil and E. O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Information and Control*, 22:132–138, 1973.

[8] P. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[9] M.M. Syslo. Transitive closure of a graph. *Zastosow. Matem.*, 14:477–480, 1974.

[10] M.M. Syslo and J. Dzikiewicz. Computational experiences with some transitive closure algorithms. *Computing*, 15:33–39, 1975.

[11] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[12] L. Thorelli. An algorithm for computing all paths in a graph. *BIT*, 6:347–349, 1966.

[13] D. Walker. Introduction to a calculus of communicating systems. Technical Report ECS-LFCS-87-22, University of Edinburgh, Laboratory for Foundations of Computer Science, 1987.

[14] S. Warshall. A theorem on boolean matrices. *J.ACM*, 9(1):11–12, January 1962.

A Implementation of Warshall's algorithm

A.1 Version wa

```
fun transcl (init,graph) =
let
    fun length' (a: 'a state list) = length a

    val sz = (length' (graph)) - 1

    val i = ref 0

    val r = ref 0

    val k = ref 0

    val statearray = array(sz+1, !init)

    fun inits (s as (S{id,suc,tauarr,...})) = (
        update(statearray,id,s);
        tauarr := array(sz+1,false);
        map (fn (S{id,...}) => update(!tauarr,id,true)) (tausfrom (!suc)))

    fun afters (S{suc,taus,tauarr,...}) = (
        taus := nil;
        i :=0;
        while (!i) <= sz do (
            if (!tauarr) sub (!i)
            then taus := (statearray sub (!i))::(!taus)
            else ();
            inc i);
        if (!taus) <> []
        then (tauref (!suc)) := (!taus)
        else ())

    fun tau_edge (S{tauarr,...}) i = (!tauarr sub (!i))

    fun add_taus (S{tauarr = arr1,...}) (S{tauarr = arr2,...}) = (
        k := 0;
        while !k <= sz do (
            if (not (!arr1 sub !k)) andalso (!arr2 sub !k)
            then update(!arr1,!k,true)
            else ();
            inc k))

in (
    map inits graph;
    i := 0;
    while !i <= sz do (
        r := 0;
        while !r <= sz do (
            if tau_edge (statearray sub (!r)) i
            then add_taus (statearray sub (!r)) (statearray sub (!i))
```

```
            else ();                                                    50
            inc r);
        inc i);
    map afters graph;
    (init,graph))
end
```

A.2 Version wl

```
fun transcl (init,graph) =
let
   fun length' (a: 'a state list) = length a

   val sz = (length' (graph)) - 1
   val i = ref 0

   val statearray = array(sz+1, !init)

   fun merge nil ys = ys                                                    10
    | merge xs nil = xs
    | merge ((x as (S{id = xid,...}))::xs) ((y as (S{id = yid,...}))::ys) =
      if xid = yid then x :: merge xs ys
      else if xid < yid
         then y :: merge (x::xs) ys
         else x :: merge xs (y::ys)

   fun sort nil ys = ys
    | sort (x::xs) ys = sort xs (merge [x] ys)
                                                                            20
   fun inits (s as (S{id,suc,taus,...})) = (
      update(statearray,id,s);
      taus := sort (tausfrom (!suc)) nil)

   fun afters (S{suc,taus,...}) =
      if (!taus) <> []
      then (tauref (!suc)) := (!taus)
      else ()

   fun tau_edge nil y = false                                               30
    | tau_edge ((S{id,...})::xs) (y:int) =
      if id = y then true
      else if id < y
         then false
         else tau_edge xs y

   fun add_taus (S{id,taus = itaus,...}) (S{taus = rtaus,...}) =
      if tau_edge (!rtaus) id
      then rtaus := merge (!rtaus) (!itaus)
      else()                                                                40

in (
   map inits graph;
   i := 0;
   while !i <= sz do (
      map (add_taus (statearray sub (!i))) graph;
      inc i);
   map afters graph;
   (init,graph))
end                                                                          50
```

B Implementation of Eve and Kurki-Suonio's algorithm

B.1 Version ea

```
fun transcl (init,graph) =
let
    fun length' (a: 'a state list) = length a

    val sz = (length' (graph)) - 1

    val statearray = array(sz+1, !init)

    val listindex = ref(sz + 1)
    val stacktop = ref ~1                                          10
    val i = ref 0
    val j = ref 0

    val vertices = array(sz+1,~1)
    val index = array(sz+1,~1)

    fun insert x nil = (x::nil)
     | insert (x as (S{id = xid,...})) ((y as (S{id = yid,...}))::ys) =
       if xid < yid then (y :: insert x ys)
       else if xid = yid then (y::ys)                              20
       else (x :: (y :: ys))

    fun sort nil ys = ys
     | sort (x::xs) ys = sort xs (insert x ys)

    fun inits (s as (S{id,suc,tauarr,taus,...})) = (
        update(statearray,id,s);
        tauarr := array(sz+1,false);
        taus := sort (tausfrom (!suc)) nil)
                                                                   30
    fun afters (S{suc,taus,tauarr,...}) = (
        taus := nil;
        i :=0;
        while (!i) <= sz do (
            if (!tauarr) sub (!i)
            then taus :=  (statearray sub (!i))::(!taus)
            else ();
            inc i);
        if (!taus) <> []
        then (tauref (!suc)) := (!taus)                            40
        else ())

    exception Empty_list;

    fun car nil = raise Empty_list
     | car (x::xs) = x
```

```
    fun cdr nil = nil
     | cdr (x::xs) = xs
                                                                          50
    fun add_taus (S{tauarr = arr1,...}) (S{tauarr = arr2,...}) =
    let
        val k = ref 0
    in (
        k := 0;
        while !k <= sz do (
            if (not (!arr1 sub !k)) andalso (!arr2 sub !k)
            then update(!arr1,!k,true)
            else ();
            inc k))                                                       60
    end

    fun closure (x as (S{id,taus,tauarr,...})) =
    let
        val w_list = ref[(!init)]
        val w = ref(!init)
        val w_id = ref 0
    in (
        inc stacktop;
        update(index,id,!stacktop);                                       70
        update(vertices,!stacktop,id);
        w_list := !taus;
        while (!w_list) <> nil do (
            w := car(!w_list);
            w_id := sid(!w);
            update(!tauarr,!w_id,true);
            if (index sub (!w_id)) = ~1
            then closure (!w)
            else ();
            if (index sub (!w_id)) <= (!stacktop)                         80
            then update(index,id,min((index sub id),(index sub (!w_id))))
            else add_taus x (statearray sub (vertices sub (index sub (!w_id))));
            w_list := cdr(!w_list));
        if id = (vertices sub (index sub id))
        then (
            dec listindex;
            w_id := (vertices sub (!stacktop));
            w := (statearray sub (!w_id));
            dec stacktop;
            update(index,!w_id,!listindex);                               90
            while (!w_id) <> id do (
                add_taus x (!w);
                w_id := (vertices sub (!stacktop));
                w := (statearray sub (!w_id));
                dec stacktop;
                update(index,!w_id,!listindex));
            update(vertices,!listindex,id))
        else ())
    end
                                                                          100
in (
```

```
    map inits graph;
    i := 0;
    while (!i) <= sz do (
        if (index sub (!i)) = ~1
        then closure (statearray sub (!i))
        else ();
        inc i);
    i := 0;
    while (!i) <= sz do (                                                    110
        j := (vertices sub (index sub (!i)));
        if (!i) <> (!j)
        then add_taus (statearray sub (!i)) (statearray sub (!j))
        else ();
        inc i);
    map afters graph;
    (init,graph))
end
```

B.2 Version el

```
fun transcl (init,graph) =
let
    fun length' (a: 'a state list) = length a

    val sz = (length' (graph)) - 1
    val statearray = array(sz+1, !init)

    val listindex = ref(sz + 1)
    val stacktop = ref ~1
    val i = ref 0                                                    10
    val j = ref 0

    val vertices = array(sz+1,~1)
    val index = array(sz+1,~1)

    fun merge nil ys = ys
     | merge xs nil = xs
     | merge ((x as (S{id = xid,...}))::xs) ((y as (S{id = yid,...}))::ys) =
       if xid = yid then x :: merge xs ys
       else if xid < yid then y :: merge (x::xs) ys                  20
          else x :: merge xs (y::ys)

    fun sort nil ys = ys
     | sort (x::xs) ys = sort xs (merge [x] ys)

    fun inits (s as (S{id,suc,newtaus,taus,...})) = (
       update(statearray,id,s);
       newtaus := nil;
       taus :=sort (tausfrom (!suc)) nil)
                                                                     30
    fun afters (S{suc,taus,newtaus,...}) =
       if (!newtaus) <> []
       then (
          taus := (!newtaus);
          (tauref (!suc)) := (!taus))
       else ()

    exception Empty_list;

    fun car nil = raise Empty_list                                   40
     | car (x::xs) = x

    fun cdr nil = nil
     | cdr (x::xs) = xs

    fun add_taus (S{newtaus = n1,...}) (S{newtaus = n2,...}) =
       n1 := merge (!n1) (!n2)

    fun closure (x as (S{id,taus,newtaus,...})) =
    let                                                              50
       val w_list = ref[(!init)]
```

```
        val w_id = ref 0
        val w = ref(!init)
    in (
        inc stacktop;
        update(index,id,!stacktop);
        update(vertices,!stacktop,id);
        w_list := !taus;
        while (!w_list) <> nil do (
            w := car(!w_list);                                                    60
            w_id := sid(!w);
            newtaus := (merge (!newtaus) [(!w)]);
            if (index sub (!w_id)) = ~1
            then closure (!w)
            else ();
            if (index sub (!w_id)) <= (!stacktop)
            then update(index,id,min((index sub id),(index sub (!w_id))))
            else add_taus x (statearray sub (vertices sub (index sub (!w_id))));
            w_list := cdr(!w_list));
        if id = (vertices sub (index sub id))                                     70
        then (
            dec listindex;
            w_id := (vertices sub (!stacktop));
            w := (statearray sub (!w_id));
            dec stacktop;
            update(index,!w_id,!listindex);
            while (!w_id) <> id do (
                add_taus x (!w);
                w_id := (vertices sub (!stacktop));
                w := (statearray sub (!w_id));                                    80
                dec stacktop;
                update(index,!w_id,!listindex));
            update(vertices,!listindex,id))
        else ())
    end

in (
    map inits graph;
    i := 0;
    while (!i) <= sz do (                                                         90
        if (index sub (!i)) = ~1
        then closure (statearray sub (!i))
        else ();
        inc i);
    i := 0;
    while (!i) <= sz do (
        j := (vertices sub (index sub (!i)));
        if (!i) <> (!j)
        then add_taus (statearray sub (!i)) (statearray sub (!j))
        else ();                                                                  100
        inc i);
    map afters graph;
    (init,graph))
end
```

C Test results on sun 3/110

Table 5 - 9 shows the execution time in seconds on a sun 3/110 machine, with 16 Mb memory.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n[1]	42.5640	18.6240	22.7040	1.3839	21.2600	0.0480	0.8640
	g[2]	5.1280	0.2720	4.8200	0.0999	4.7080	0.0000	0.0360
	t[3]	47.6919	18.8960	27.5240	1.4840	25.9679	0.0480	0.9000
wa	n	44.3440	18.6640	24.4560	0.7800	23.6080	0.0480	0.8640
	g	4.4000	0.2839	1.3879	0.0400	1.3480	0.0000	0.0440
	t	48.7439	18.9480	25.8440	0.8200	24.7759	0.0480	0.9079
ea	n	41.9159	18.7680	21.8839	0.4280	21.3920	0.0440	0.8720
	g	4.8719	0.2000	1.3160	0.0400	1.2760	0.0000	0.6839
	t	46.7880	18.7880	23.1999	0.4680	22.6679	0.0440	1.5559
wl	n	43.6359	18.6839	23.7199	0.3040	23.3560	0.0520	0.8640
	g	1.7079	0.2880	1.3480	0.0000	1.3480	0.0000	0.0440
	t	45.3440	18.7920	25.0680	0.3040	24.7040	0.0520	0.9080
el	n	41.1600	18.7639	21.1479	0.0640	21.0320	0.0480	0.8760
	g	1.5720	0.1840	1.3599	0.0000	1.3599	0.0000	0.0279
	t	42.7320	18.7680	22.5079	0.0640	22.3920	0.0480	0.9040

Table 5: Execution time for the agent User-Timers.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	49.5559	21.2280	24.3999	1.7360	22.6000	0.0600	2.9440
	g	1.8959	0.2799	1.5160	0.1400	1.3679	0.0080	0.0559
	t	51.4519	21.5079	25.9160	1.8760	23.9680	0.0679	2.8200
wa	n	49.5480	21.2160	24.4160	1.6000	22.7519	0.0560	2.9479
	g	2.3439	0.8600	1.3799	0.0520	1.3279	0.0000	0.0520
	t	51.8919	22.0760	25.7959	1.6519	24.0799	0.0560	2.6400
ea	n	49.1039	21.2119	23.9759	1.1080	22.7920	0.0600	2.9240
	g	1.8359	0.2480	1.4759	0.0680	1.4079	0.0000	0.0680
	t	50.9399	21.4600	25.4520	1.1759	24.1999	0.0600	2.6319
wl	n	49.2440	21.1640	24.1600	0.4640	23.6280	0.0640	2.9400
	g	5.1959	0.2880	4.7759	0.0120	4.7560	0.0080	0.0680
	t	54.4399	21.4520	28.9360	0.4760	28.3839	0.0720	2.6479
el	n	46.6519	21.1120	21.6319	0.1040	21.4520	0.0600	2.9400
	g	5.1279	0.2680	1.4039	0.0120	1.3919	0.0000	2.0759
	t	51.7800	21.3800	22.8560	0.1160	22.8440	0.0600	5.0160

Table 6: Execution time for the agent PE.

[1] non gc time
[2] gc time
[3] total time (non gc time + gc time)

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	84.8599	71.1719	11.7679	3.3399	8.3399	0.0600	1.6839
	g	4.8319	0.9120	2.3879	0.2720	2.1160	0.0000	0.7920
	t	89.6920	72.0840	14.1560	3.6119	10.4559	0.0600	2.4760
wa	n	87.4080	71.2560	14.2359	5.7880	8.3640	0.0640	1.6839
	g	1.6079	0.8240	0.7400	0.0720	0.6679	0.0000	0.0360
	t	89.0159	72.0800	14.9759	5.8600	9.0320	0.0640	1.7200
ea	n	83.7720	70.9759	10.8719	2.3439	8.4559	0.0600	1.6800
	g	1.6399	0.8720	0.7360	0.0880	0.6399	0.0080	0.0280
	t	85.4120	71.8479	11.6080	2.4319	9.0960	0.0680	1.7079
wl	n	82.5159	71.0320	9.5399	0.9920	8.4839	0.0640	1.6960
	g	1.5200	0.7960	0.6679	0.0520	0.6160	0.0000	0.0520
	t	84.0360	71.8280	10.2080	0.8640	9.0999	0.0640	1.7480
el	n	81.7839	71.1719	8.6880	0.2799	8.3319	0.0640	1.6759
	g	1.5160	0.7880	0.6600	0.0080	0.6519	0.0000	0.0520
	t	83.2999	71.9599	9.3479	0.2880	8.9840	0.0640	1.7280

Table 7: Execution time for the agent PE-Respnet.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	211.8760	47.5799	161.3360	10.1720	151.0360	0.1160	2.5680
	g	10.8679	0.4160	10.3680	0.7240	9.6239	0.0200	0.0840
	t	222.7439	47.8159	171.7040	10.8960	160.6599	0.1359	2.6520
wa	n	224.8599	47.6840	174.2479	4.5000	169.6080	0.1119	2.5479
	g	10.2639	0.3680	9.8279	0.1679	9.6239	0.0360	0.0680
	t	235.1240	47.8719	184.0760	4.6680	179.2319	0.1480	2.6160
ea	n	205.1320	47.7320	154.4599	2.1920	152.1280	0.1080	2.5560
	g	10.4160	0.4520	9.9160	0.2160	9.6639	0.0360	0.0480
	t	215.5480	48.1839	164.3760	2.4079	161.7919	0.1440	2.6040
wl	n	220.2560	47.5840	169.6999	1.5680	168.0000	0.1200	2.5840
	g	10.0599	0.4320	9.5520	0.0200	9.5000	0.0320	0.0759
	t	230.3160	48.0160	179.2520	1.5879	177.5000	0.1520	2.6600
el	n	201.6400	47.7359	150.9280	0.1399	150.6559	0.1160	2.5800
	g	9.9439	0.3760	9.4719	0.0000	9.4719	0.0000	0.0640
	t	211.5840	48.1120	160.4000	0.1399	160.1279	0.1160	2.6440

Table 8: Execution time for the agent Net-Timers.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	1726.5560	543.8160	1160.2239	501.1399	658.4120	0.6599	22.4119
	g	108.0519	7.8599	99.9279	43.2960	56.5920	0.0400	0.2640
	t	1834.6080	551.4959	1260.1520	544.4360	715.0040	0.7000	22.6759
wa	n	1889.6839	544.8280	1322.3319	638.2960	683.3199	0.7040	22.3960
	g	81.5519	7.7999	73.4840	17.1999	56.2840	0.0000	0.2680
	t	1971.2360	552.6279	1395.8160	655.4960	739.6040	0.7040	22.6640
ea	n	1362.1680	546.5399	793.1160	104.5159	687.8999	0.6919	22.3879
	g	71.7759	7.7359	63.7960	19.6840	43.9720	0.1399	0.2440
	t	1433.9439	554.2760	856.9119	124.2000	731.8720	0.8320	22.6319
wl	n	1306.9199	543.3199	741.1240	45.3999	695.0080	0.7000	22.3399
	g	73.5120	7.9360	65.2560	13.1919	52.0039	0.0600	0.3160
	t	1380.4320	551.2560	806.3799	58.5919	747.0119	0.7600	22.6559
el	n	1236.2000	544.4439	669.2479	1.6839	666.8520	0.7000	22.3920
	g	66.0080	7.6760	58.0199	5.8280	52.1679	0.0240	0.3119
	t	1302.2080	552.1200	727.2679	7.5120	719.0200	0.7240	22.7040

Table 9: Execution time for the agent Pro.

D Test results on sun 4/60

Table 10 - 14 shows the execution time in seconds on a sun 4/60 machine, with 8 Mb memory.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	14.8039	5.0099	9.3899	0.5599	8.8060	0.0180	0.2960
	g	0.9200	0.0720	0.8299	0.0300	0.8000	0.0000	0.0180
	t	15.7240	5.0820	10.2199	0.5900	9.6059	0.0180	0.3140
wa	n	14.5719	5.0079	9.1600	0.3399	8.7980	0.0200	0.2940
	g	0.9040	0.0820	0.8039	0.0180	0.7859	0.0000	0.0120
	t	15.4759	5.0900	9.9640	0.3580	9.5839	0.0200	0.3060
ea	n	14.3460	4.9939	8.9439	0.1760	8.7439	0.0200	0.2940
	g	0.8660	0.0700	0.7820	0.0000	0.7820	0.0000	0.0140
	t	15.2119	5.0640	9.7260	0.1760	9.5259	0.0200	0.3080
wl	n	14.3980	5.0039	8.9800	0.1440	8.8159	0.0180	0.2960
	g	0.8820	0.0740	0.7900	0.0000	0.7900	0.0000	0.0180
	t	15.2800	5.0780	9.7699	0.1440	9.6059	0.0180	0.3140
el	n	14.2740	4.9940	8.8779	0.0280	8.8279	0.0200	0.2940
	g	0.8520	0.0679	0.7700	0.0000	0.7700	0.0000	0.0140
	t	15.1259	5.0620	9.6479	0.0280	9.5980	0.0200	0.3080

Table 10: Execution time for the agent User-Timers.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	16.6559	6.0039	9.3360	0.7140	8.5979	0.0200	1.0080
	g	3.1260	2.2459	0.8380	0.0480	0.7900	0.0000	0.0300
	t	19.7820	8.2500	10.1739	0.7620	9.3879	0.0200	0.8580
wa	n	16.3920	5.8480	9.2620	0.5880	8.6539	0.0200	0.9780
	g	3.1060	2.2480	0.7960	0.0220	0.7739	0.0000	0.0380
	t	19.4979	8.0960	10.0579	0.6100	9.4279	0.0200	1.0160
ea	n	16.1260	5.8340	8.9640	0.4080	8.5300	0.0200	1.0280
	g	3.0520	2.2380	0.7840	0.0160	0.7680	0.0000	0.0300
	t	19.1780	8.0719	9.7480	0.4240	9.2979	0.0200	0.8780
wl	n	16.3379	5.8379	9.2119	0.2220	8.9659	0.0180	0.9880
	g	0.9040	0.0800	0.7960	0.0040	0.7920	0.0000	0.0200
	t	17.2419	5.9180	9.8279	0.2260	9.7579	0.0180	0.6480
el	n	16.1319	5.8040	9.0540	0.0500	8.9840	0.0200	0.9800
	g	0.9240	0.0960	0.7880	0.0000	0.7880	0.0000	0.0300
	t	16.8759	5.9000	9.8420	0.0500	9.7720	0.0200	1.0100

Table 11: Execution time for the agent PE.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	25.0560	19.7839	4.5820	1.3240	3.2299	0.0220	0.6199
	g	0.6659	0.2040	0.4440	0.1440	0.3000	0.0000	0.0180
	t	25.7220	19.9879	5.0259	1.4680	3.5299	0.0220	0.6380
wa	n	25.4400	19.4260	5.3119	2.0119	3.2720	0.0200	0.6280
	g	0.5760	0.2239	0.3259	0.0139	0.3120	0.0000	0.0260
	t	26.0160	19.6499	5.6379	2.0259	3.5839	0.0200	0.6540
ea	n	23.9819	19.2160	4.0739	0.8380	3.2120	0.0200	0.6260
	g	0.5860	0.2060	0.3500	0.0360	0.3140	0.0000	0.0139
	t	24.5679	19.4220	4.4239	0.8740	3.5260	0.0200	0.6399
wl	n	23.7860	19.3759	3.7339	0.4640	3.2380	0.0240	0.6100
	g	2.7400	2.3820	0.3479	0.0040	0.3379	0.0060	0.0100
	t	26.5259	21.7579	4.0819	0.4680	3.5760	0.0300	0.6200
el	n	24.5940	20.5560	3.3580	0.1240	3.2079	0.0220	0.6060
	g	2.8119	2.4399	0.3359	0.0000	0.3359	0.0000	0.0360
	t	27.4059	22.9960	3.6939	0.1240	3.5440	0.0220	0.6420

Table 12: Execution time for the agent PE-Respnet.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	82.1020	13.7900	67.2899	4.0759	63.1679	0.0400	0.9060
	g	6.3199	0.1540	6.1379	0.3580	5.7799	0.0000	0.0280
	t	88.4219	13.9439	73.4280	4.4340	68.9480	0.0400	0.9340
wa	n	80.2899	14.0680	65.2139	1.9140	63.2539	0.0420	0.8920
	g	6.0060	0.1440	5.8239	0.0760	5.7480	0.0000	0.0320
	t	86.2960	14.2119	71.0379	1.9899	69.0019	0.0420	0.9239
ea	n	78.5639	13.6500	63.8999	0.9719	62.8840	0.0400	0.8980
	g	5.9820	0.1380	5.8280	0.0960	5.7320	0.0000	0.0160
	t	84.5460	13.7880	69.7279	1.0680	68.6159	0.0400	0.9140
wl	n	78.9800	13.8460	64.1059	0.7739	63.2860	0.0440	0.9040
	g	5.9300	0.1440	5.7679	0.0060	5.7619	0.0000	0.0180
	t	84.9099	13.9900	69.8739	0.7800	69.0480	0.0440	0.9220
el	n	78.2399	13.7979	63.4260	0.0640	63.3159	0.0400	0.9000
	g	5.9299	0.1400	5.7579	0.0080	5.7500	0.0000	0.0320
	t	84.1700	13.9379	69.1839	0.0720	69.0660	0.0400	0.9320

Table 13: Execution time for the agent Net-Timers.

		minimize	mkgraph	obscl	transcl	actcl	reflexcl	Equiv.mi
old	n	620.6159	158.4200	451.5359	185.3679	265.8480	0.3120	10.6219
	g	58.1480	3.6819	54.3339	20.8440	33.4899	0.0000	0.1320
	t	678.7639	162.1020	505.8700	206.2119	299.3380	0.3120	10.7540
wa	n	660.7379	158.9759	491.7299	226.4879	264.9119	0.3140	10.0000
	g	43.5199	3.6680	39.8079	6.1079	33.5720	0.1280	0.0440
	t	704.2580	162.6440	531.5379	232.5960	298.4839	0.4420	10.0440
ea	n	478.0820	158.7299	309.0079	37.7920	270.8939	0.3160	10.2980
	g	42.8999	3.6339	39.0860	10.9059	28.1799	0.0000	0.1800
	t	520.9819	162.3640	348.0940	48.6980	299.0740	0.3160	10.2980
wl	n	455.3079	158.5900	286.9440	23.6699	262.9959	0.2740	9.7340
	g	39.0380	3.6640	35.2740	4.0979	31.1299	0.0460	0.1000
	t	494.3460	162.2539	322.2179	27.5880	294.1259	0.3200	9.8340
el	n	442.0140	159.3320	273.2040	0.7660	272.1580	0.2719	9.4319
	g	34.3519	3.6040	30.6540	0.2120	30.4059	0.0360	0.0940
	t	476.3659	162.9359	303.8580	0.9780	302.5639	0.3080	9.5259

Table 14: Execution time for the agent Pro.